Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance Through Transpilation and Type Hints

Mateusz Bysiek, Aleksandr Drozd, Satoshi Matsuoka

Tokyo Institute of Technology

bysiek.m.aa@m.titech.ac.jp, alex@smg.is.titech.ac.jp, matsu@is.titech.ac.jp

6th Workshop on Python for High-Performance and Scientific Computing, 14 November 2016

Two-way translation, exhibit 1

English ↔ Japanese

A game. Open Google Translate. Enter a phrase in language A, translate it to language B, translate the result to language A, translate the latest result to language B, and so on until you encounter anything you've seen before.

I just called to say I love you.
愛しているというために電話しただけさ。
I just called to say I love you.
...

— according to Google Translate (`https://translate.google.com/`) on 8 November 2016.

Two-way translation, exhibit 2

English ↔ Japanese

I can’t get no satisfaction.
私は満足を得ることはできません。
I can not get satisfied.
私は満足して取得傾けます。
I cant get it pleased.
私はそれを喜んで取得することはできません。
I can not get pleased with it.
私はそれに満足して取得することはできません。
I can not get pleased with it.
...

— according to Google Translate (`https://translate.google.com/`) on 8 November 2016.

Two-way translation, exhibit 3

English ↔ French

I can't get no satisfaction.
Je ne peux pas obtenir aucune satisfaction.
I can not get no satisfaction.
Je peux pas obtenir satisfaction.
I can not get No Satisfaction.
Je peux pas obtenir satisfaction.
...

— according to Google Translate (`https://translate.google.com/`) on 8 November 2016.

Two-way translation, exhibit 4

Fortran 77/90/95 ↔ Python 3

Not available.

— according to Google (`https://google.com/`) on 8 November 2016.

But why bother?

Because Python is super-useful and Fortran is super-fast. If only we could:

- Take Fortran, and manipulate it as if it was (maintainable) Python.
- Take Python, and run it as if it was (super fast) Fortran.

Then we could create maintainable HPC solutions that build upon decades of Fortran's legacy and achieve its top performance.

How to make Python applications faster?

We have solutions for taking Python and running it faster.

Numba [1]

JIT compiler on NumPy-enabled Python code.

- Uses type inference.
- Doesn't address all performance issues.

Cython [2]

Compiler framework and language derived from Python.

- Uses own variable declaration syntax.
- Not code-compatible with Python.
- Doesn't address all performance issues.

How to make Fortran more maintainable?

Do we have solutions for taking Fortran and making it easier to manage?

f2py [3]

Triggers AOT compilation of existing Fortran code and creates a Python interface for it.

- Doesn't migrate Fortran code.
- Requires care for correct use.
- Doesn't really address the maintainability issue.

Making the workflow possible – techniques used

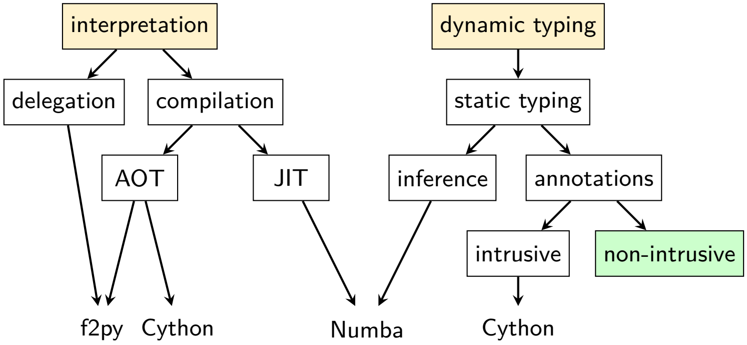

Figure: Performance issues and translation feasibility issues in Python (marked in yellow) and an approach not covered by existing solutions (marked in green) called *type hints*.

Type hints in Python

Introduced in Python 3.5 (PEPs 483 [4] and 484 [5]) but backwards-compatible.
Provided as comments or annotations:

- comments in Python 2 and onwards,
- function annotations since 3.0 (PEP 3107 [6]), and
- variable annotations since 3.6 (PEP 526 [7]).

Not using hints:
Python 2 & 3

```
def inc(a):
    c = 1
    return a + c
```

Using hints in different styles (i.e. type comments and/or annotations):

Python 2+

```
def inc(a):
    # type: (int) -> int
    c = 1 # type: int
    return a + c
```

Python 3.0+

```
def inc(a: int) -> int:

    c = 1 # type: int
    return a + c
```

Python 3.6+

```
def inc(a: int) -> int:

    c: int = 1
    return a + c
```

Recipe for HPC-enabled Fortran → Python migration

...that relies on two-way Fortran ↔ Python translation

We need:

1. Ability to express static typing in Python,
2. ability to inspect and alter Python at runtime,
3. well-defined mapping between reasonable subsets of Fortran and Python,
4. human-oriented transpiler (i.e. source-to-source compiler) that uses that mapping,
5. ability to execute Fortran from within Python,
6. a user-friendly framework that orchestrates all of the above.

We already have 1. type hints, 2. simply Python and 5. f2py.

The 3. , 4. and 6. were not available until now.

Contributions

We supply those missing ingredients:

1. Python-Fortran two-way transpiler that generates human-readable output.
   - It handles basic types, syntax, basic array manipulation, selected idioms, and a part of MPI. Also doesn't ignore code-embedded documentation and/or comments.
   - Fortran versions: 77, 90 and 95; Python 3.
2. Workflow design for migration of legacy Fortran applications to Python without sacrificing their performance.
   - As a sub-workflow, a workflow to boost performance of Python applications to the level of their Fortran equivalents.

Workflow for migrating legacy Fortran to Python

...without sacrificing performance

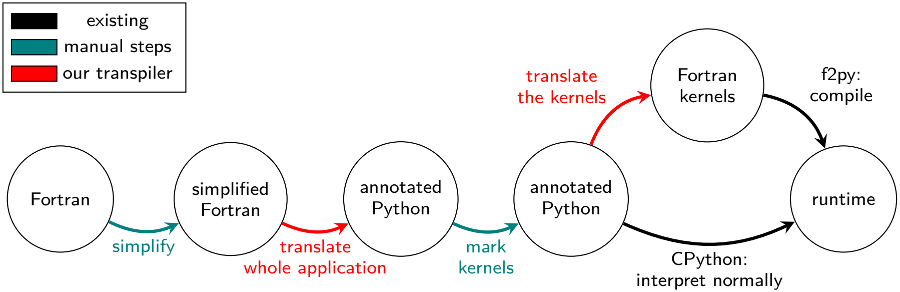

Figure: Translation of Fortran to Python creates Fortran-like (i.e. compatible with static type system) Python code.

Sub-workflow for boosting Python's performance

...to the level equivalent of manual transaltion to Fortran

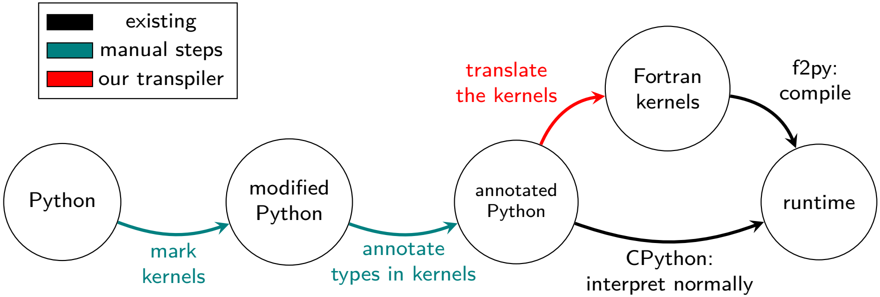

Figure: Performance of Fortran-like Python code can be enhanced by leveraging the similarities between subsets of Python and Fortran.

Evaluation environment

Ubuntu 14.04.1 x64, Linux kernel 4.2.0-41, Python 3.5.1.

ATLAS	3.10.1-4
LAPACK	3.5.0-1
numpy	1.11.1
llvmlite	0.12.0
numba	0.26.0

Table: Versions of packages installed in evaluation environment.

APU	AMD A10-5800K
CPU clock	3.8 GHz $\sim$ 4.2 GHz
CPU cores	4
main memory	16 GB at 800 MHz

Table: Hardware information for the evaluation environment.

Boosting Python: Matrix multiplication

Single-threaded DGEMM in 4 versions:

- baseline: pure Python;
- JIT-compiled pure Python, using Numba;
- Fortran implementation, interfaced with Python using f2py;
- pure Python with type hints, JIT-translated using our transpiler.

Pseudocode:

```
def my_matmul(a, b, a_width, a_height, b_width):
    c = [0 for _ in range(b_width * a_height)]
    for y in range(a_height):
        for i in range(a_width):
            for x in range(b_width):
                c[y * b_width + x] += \
                    a[y * a_width + i] * b[i * b_width + x]
    return c
```

Matrix multiplication – compilation/translation overhead

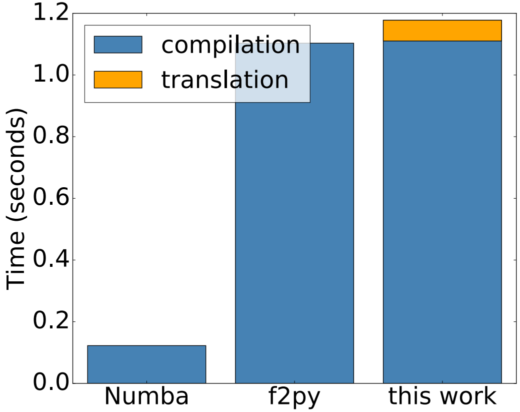

Figure: Compilation and/or translation overhead comparison for existing approaches and this work. Numba JIT-compiles completely in memory, whereas f2py and we (because we use f2py as part of the workflow) use intermediate files and create a shared library file.

Matrix multiplication – compiled binary reuse

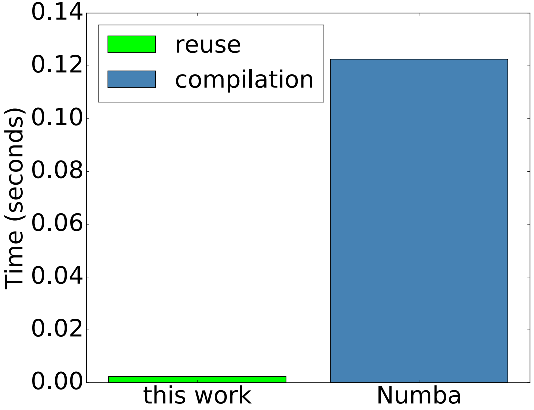

Figure: Binary object reuse in our framework and Numba's compilation time. Numba needs to recompile with each application launch, while we can simply load the compiled file.

Matrix multiplication – computational results

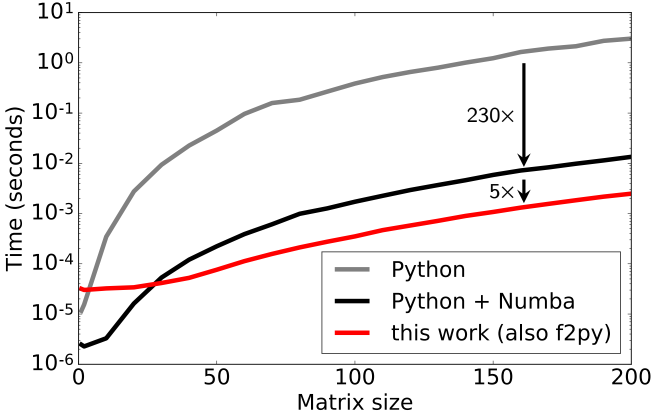

Figure: Computational performance of pure Python while using our framework is the same as launching Fortran implementation through f2py.

Migrating Fortran: Miranda IO

Characteristics:

- I/O benchmarking application – I/O kernel extracted from Miranda, a weapons simulation application from LLNL;
- pure Fortran – mostly 77, plus few 95' intrinsic functions;
- uses MPI;
- around 200 lines of code.

Process of migration:

- minor refactoring of Fortran before automatic translation to Python;
- minor refactoring of resulting Python to extract the kernel to a separate function and mark it for translation.

Miranda IO – compilation/translation overhead

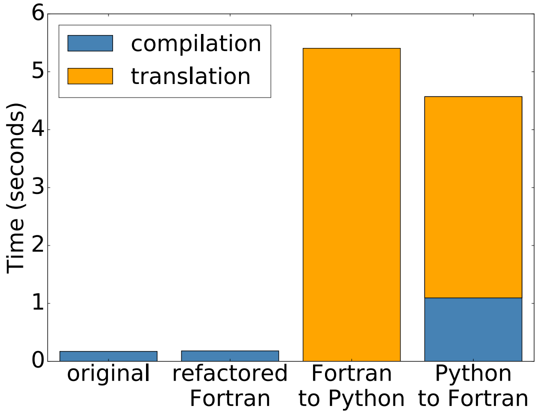

Figure: Translation and compilation overhead comparison – original Miranda IO, refactored version, translation to Python and reexpression to Fortran. Compilation slowdown from original to dynamically created Fortran: 25x.

Miranda IO – computational results

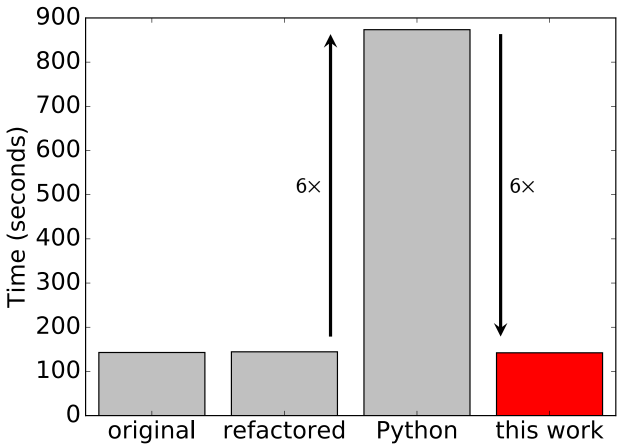

Figure: Our framwork completely recovers from computational performance drop of Python version of Miranda IO while maintaining the code in Python.

Defining the mapping – types

Fortran	**Python**
Basic types:	
`logical`	`bool`
`integer`	`int`
`integer*4`	`np.int32 # import numpy as np`
`integer*8`	`np.int64`
`real`	`float`
`real*4`	`np.single`
`real*8`	`np.double`
`character`	`str`
`character*1024`	`str`
More complex declarations:	
`integer*8, dimension(0:n - 1)`	`np.ndarray((n,), dtype=np.int64)`
`integer*4, dimension(0:n - 1, 0:k - 1)`	`np.ndarray((n, k), dtype=np.int32)`

When transpiling, we adopt type hints style of Python 3.0-3.5.

Defining the mapping – examples of supported syntax

In Fortran...

... variables are declared:

```
integer*8 :: A(0:n - 1)
```

... array operations look like:

```
B(0, 0, :) = A(:)
```

... copy on assignment is implicit:

```
C = A
```

... loops have strict form:

```
do i = 0, n - 1
    A(i) = i
end do
```

In Python...

... no declarations needed:

```
A = np.zeros(n, dtype=np.int64)
```

... they can look just the same:

```
B[0, 0, :] = A[:]
```

... explicit copy is needed:

```
C = A.copy()
```

... loops can look alike:

```
for i in range(n):
    A[i] = i
```

Defining the mapping – examples of supported syntax

In Fortran...

... including external APIs:

```
# arrays are built-in
include 'mpif.h'
use mpi
```

... APIs are traditional:

```
call MPI_COMM_SIZE(
    MPI_COMM_WORLD, n, err)
```

In Python...

... uses different idioms:

```
import numpy as np
import mpi4py
from mpi4py import MPI
```

... APIs are object-oriented:

```
n = MPI.COMM_WORLD.Get_size()
# error variable: err
```

Defining the mapping – examples of unsupported syntax

Fortran has...

... n-dimensional assignment expression:

```
forall(i=1:ni,j=1:nj) B(i, j) = i * j
```

... pragmas for compiler extensions:

```
!$acc loop collapse(4) independent gang vector(32)
!$OMP PARALLEL
do ...
```

In Python...

... variables can change type:

```
n = 0
n = 'Tokodai'
```

... there are custom types:

```
class MyClass(BaseClass):
    ...
```

Conclusion

Maintainability, extensibility and interoperability can be improved without sacrificing performance.

- Performance of DGEMM in Python equals that of Fortran:
  - there is no computational overhead;
  - Python code can be as fast as Fortran.
- Benchmark migrated to Python retains original performance.
  - Two-way translation approach is not only feasible, but also useful.

We can pick and choose: the best features of Python, and the best features of Fortran.

Bibliography

[1] Siu Kwan Lam, Antoine Pitrou and Stanley Seibert. 'Numba: A LLVM-based Python JIT Compiler'. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: ACM, 2015, 7:1–7:6. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162. URL: http://doi.acm.org/10.1145/2833157.2833162 (cit. on p. 6).

[2] Stephan Behnel et al. *Cython: C-Extensions for Python, 2008*. URL: http://cython.org/ (cit. on p. 6).

[3] Pearu Peterson. 'F2PY: a tool for connecting Fortran and Python programs'. In: *International Journal of Computational Science and Engineering* 4.4 (2009), pp. 296–305. URL: http://cens.ioc.ee/~pearu/papers/IJCSE4.4%5C_Paper%5C_8.pdf (cit. on p. 7).

[4] Guido van Rossum and Ivan Levkivskyi. *PEP 483 – The Theory of Type Hints*. Dec. 2014. URL: https://www.python.org/dev/peps/pep-0483/ (visited on 05/09/2016) (cit. on p. 9).

[5] Guido van Rossum, Jukka Lehtosalo and Łukasz Langa. *PEP 484 – Type Hints*. Sept. 2014. URL: https://www.python.org/dev/peps/pep-0484/ (visited on 10/07/2016) (cit. on p. 9).

[6] Collin Winter and Tony Lownds. *PEP 3107 – Function Annotations*. Dec. 2006. URL: https://www.python.org/dev/peps/pep-3107/ (visited on 07/11/2016) (cit. on p. 9).

[7] Ryan Gonzalez et al. *PEP 526 – Syntax for Variable Annotations*. Sept. 2016. URL: https://www.python.org/dev/peps/pep-0526/ (visited on 07/11/2016) (cit. on p. 9).

Next steps

Overcoming scope limitations:

- go beyond current restricted subset of Fortran and Python;
- support other languages.

Towards performance-portable HPC Python:

- combine transpilation with code generation.

Resolving implementation issues:

- improve compilation and translation speed.

Matrix multiplication – Python without type hints

```python
@numba.jit # enable Numba JIT compilation
def my_matmul(
        a,
        b,
        a_width, a_height,
        b_width
        ):
    c = np.zeros(b_width * a_height, dtype=np.double)
    for y in range(a_height):
        for i in range(a_width):
            for x in range(b_width):
                c[y * b_width + x] += \
                    a[y * a_width + i] * b[i * b_width + x]
    return c
```

Matrix multiplication – Python 3 with type hints

```python
@reexpress('Fortran77') # enable JIT transpilation using our work
def my_matmul(
        a: np.ndarray((200 * 200,), dtype=np.double),
        b: np.ndarray((200 * 200,), dtype=np.double),
        a_width: np.int32, a_height: np.int32,
        b_width: np.int32
        ) -> np.ndarray((200 * 200,), dtype=np.double):
    c = np.zeros(b_width * a_height, dtype=np.double)
    for y in range(a_height): # type: np.int32
        for i in range(a_width): # type: np.int32
            for x in range(b_width): # type: np.int32
                c[y * b_width + x] += \
                    a[y * a_width + i] * b[i * b_width + x]
    return c
```

Matrix multiplication – Fortran 77 equivalent

```fortran
      subroutine my_matmul_fortran(a, b, c, a_width, a_height,
     &    b_width)
      integer*4, parameter :: max_width = 200
      integer*4, parameter :: max_height = 200
      real*8, intent(in) :: a(max_width * max_height)
      real*8, intent(in) :: b(max_height * max_width)
      real*8, intent(out) :: c(max_height * max_height)
      integer*4, intent(in) :: a_width, a_height, b_width
      integer*4 :: y, i, x

      c = 0
      do y = 1, a_height
          do i = 1, a_width
              do x = 1, b_width
                  c((y - 1) * b_width + x) =
     &                c((y - 1) * b_width + x) +
     &                a((y - 1) * a_width + i) *
     &                b((i - 1) * b_width + x)
              end do
          end do
      end do
      return
      end subroutine my_matmul_fortran
```